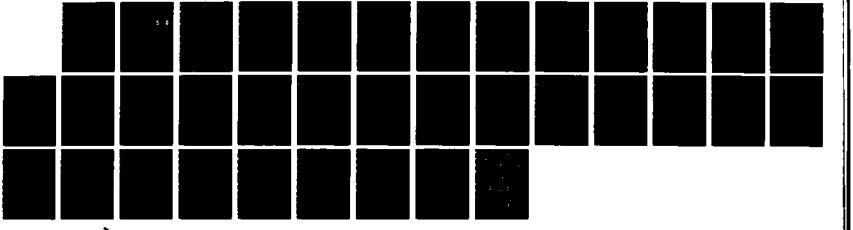
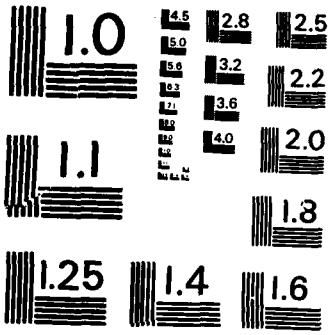
AD-A166 611 DEVELOPMENT OF A BUTTERFLY (TRADEMARK) MULTIPROCESSOR TEST BED: CONTENTION IN THE BUTTERFLY SWITCH(U) BOLT BERANEK AND NEWMAN INC CAMBRIDGE MA DEC 85 BBN-5875 1/1

UNCLASSIFIED MDA903-84-C-0033 F/G 9/2 NL





BBN Laboratories Incorporated

A Subsidiary of Bolt Beranek and Newman Inc.

Report No. 5875

Quarterly Technical Report No. 4
July 16, 1984-October 15, 1984
Development of a Butterfly Multiprocessor Test Bed
Contention in the Butterfly Switch

December 1985

Prepared for:
Defense Advanced Research Projects Agency
Engineering Applications Office

SECURITY CLASSIFICATION OF THIS PAGE

AD-A166611

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188
Exp. Date: Jun 30, 1986

1a REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Distribution Unlimited
2b DECLASSIFICATION / DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S) 5875	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Bolt Beranek and Newman Inc.	6b. OFFICE SYMBOL (If applicable)
	7a. NAME OF MONITORING ORGANIZATION Defense Advanced Research Projects Agency
6c. ADDRESS (City, State, and ZIP Code) 10 Moulton Street Cambridge, MA 02238	7b. ADDRESS (City, State, and ZIP Code) 1400 Wilson Boulevard Arlington, VA 22209
8a. NAME OF FUNDING / SPONSORING ORGANIZATION	8b. OFFICE SYMBOL (If applicable)
	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER MDA903-84-C-0033
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Development of a Butterfly Multiprocessor Test Bed: Contention in the Butterfly Switch

12. PERSONAL AUTHOR(S)

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Quarterly Technical Rpt	FROM 7/16/84 TO 10/15/84	1985, December	30

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report describes a model for contention in the Butterfly switch, and presents experimental results.

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Edward A. Starr	22b. TELEPHONE (Include Area Code) (617) 497-3307 / 22c OFFICE SYMBOL

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

Quarterly Technical Report No. 4
July 16, 1984 - October 15, 1984

DEVELOPMENT OF A BUTTERFLY™ MULTIPROCESSOR TESTBED
Contention in the Butterfly Switch

December 1985

Prepared for:

Dr. Clinton Kelly, Director
Defense Advance Research Projects Agency
Arlington, Virginia 22209

Table of Contents

FIGURES

TABLES

1. OVERVIEW

Two topics are treated in this Quarterly Technical Report. First, a model for the contention in the Butterfly[TM]* Switch is presented and discussed. Second, the results of measurements of contention using a 16-processor Butterfly machine are presented. These results substantiate the switch model, and indicate that the Butterfly Switch performs very well.

The efforts reported here were done in preparation for the tests to be conducted in the 128-processor Butterfly Test Bed in the next quarter.

*Butterfly is a trademark of Bolt Beranek and New man Inc.

2. CONTENTION IN THE BUTTERFLY SWITCH

2.1. Introduction

One of the characteristics of the Butterfly™ Switch is the possibility that two or more processors will attempt to use the same switch path. In this event, one will succeed and the others must wait their turn. This effect averaged over the course of running an entire program will result statistically in a slowdown. The purpose of this section is to derive a model of the process and to compute the expected value of the slowdown.

The value of the slowdown will depend, on the program. The critical parameter of the program is the frequency of references to global memory. Let us define the variable E to be the interval between such references to memory in the complete absence of a switch. This interval is a characteristic of the program running on an ordinary MC68000, and can be measured on a Butterfly system by running the whole program (with its data) in a single node. We want to calculate the corresponding value of the slowdown time F of the same program running on a Butterfly system with the data spread uniformly over the machine, but with all the other processors competing for the switch as though they were running the same program (or at least a similar one) - with the same value of E.

The result will depend on the machine configuration one is dealing with. Butterfly machines differ most obviously in the number of processors they have. They also differ, when the number of processors is not a power of four, in the way the switch is configured. (For powers of four there is a configuration so obvious that we have never built one of the alternatives - such alternatives could exist however).

The result will also depend on program statistics. If the pattern of reference to global memory is such that all the processors go to the same memory (or use the same path in the switch) at the same time, then the machine will slow down dramatically. Our computations assume that global references go to random addresses, and thus we compute the expected values (rather than the worst case) of the delays.

2.2. The Model

Let us first deal with two simple effects. If the global data is distributed at random over the machine, there is some rather small probability that its memory will actually be local to a single processor. In that case the memory reference is faster. This has the effect of lengthening E by the factor (P+1)/P (P is the number of processors). We assume that this effect is accounted for in E, and that "global references" are always non-local.

Next, there will be a mechanical slowdown of every non-local reference simply because it takes an otherwise empty switch 4 microseconds to make the reference. This number includes everything, and is the measured difference between a local load and a remote load of a 16-bit quantity. For unknown reasons the time of a double word load measures at 7.5 rather than 8 microseconds. Perhaps this is simple measurement noise. We will keep the delay symbolic rather than numeric, so let R indicate the mechanical slowdown of the switch, and remember that R = 4. We assume that the references are single word transactions; if they are not, one simply adjusts E accordingly.

nGlossary of Symbols

A:	Time in the absence of contention = E + R.	P:	Number of processors.
D:	Extra delay due to switch contention.	p:	Probability of collision at a single stage of the Butterfly Switch.
D_0:	Time wasted on trial where collision occurs (measured as about 2 microseconds).	q:	Probability of collision on a one way reference through the switch.
E:	Interval between references to memory in a program.	R:	Delay attributable to a mechanical slowdown (approximately 4 microseconds).
F:	Time for program to run on a Butterfly machine with shared memory.	r:	Number of ranks in the Butterfly Switch.
h:	$\frac{3r}{4}$	x:	Relative slowdown due to contention compared to the absence of contention.

Now we proceed with the model. In the absence of contention in the switch:

$$F = E + R \tag{2.1}$$

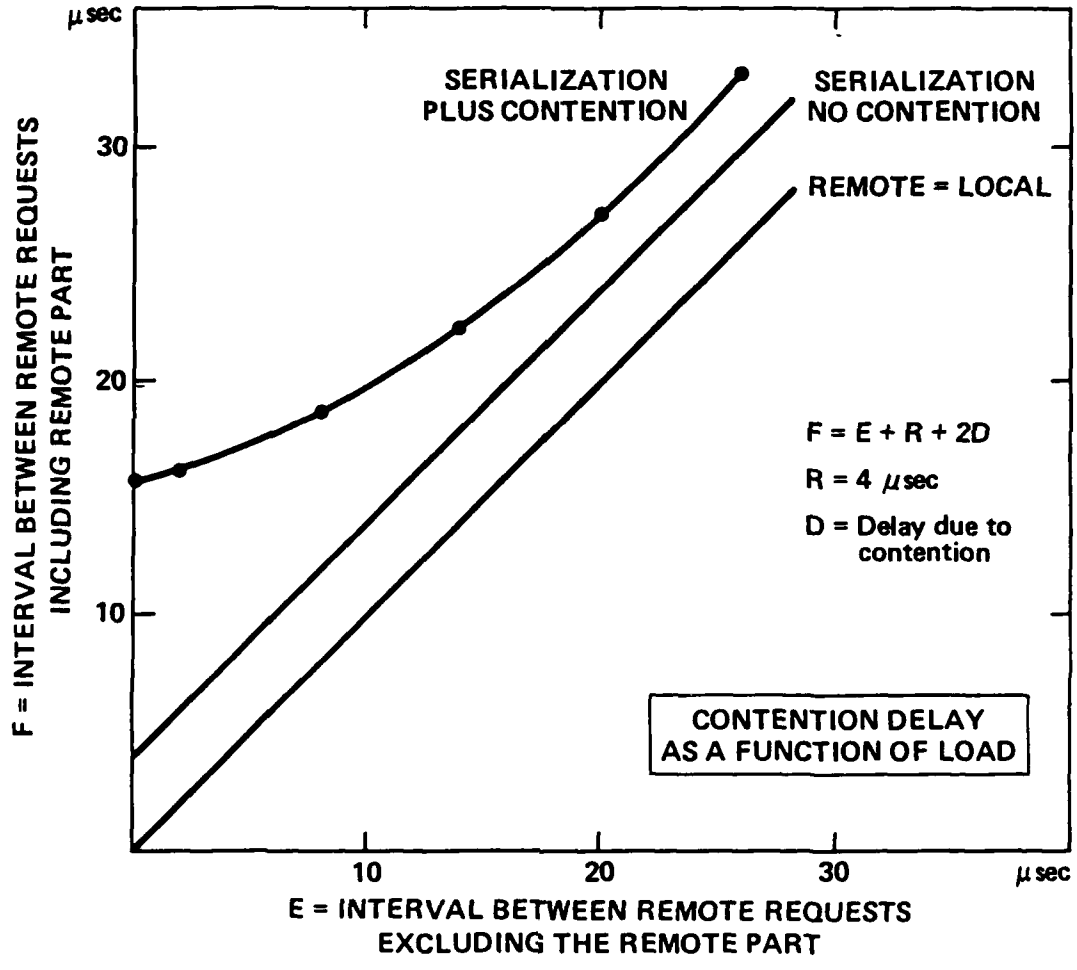
Figure 2.1 shows this simple relationship, which tends to hold true for most programs running on a Butterfly machine even in the presence of a little contention. As the period between memory references (E) gets large compared to R, E approaches F. In the presence of significant contention there is another term due to contention delay:

$$F = E + R + 2(D) \tag{2.2}$$

where D is the extra delay due to conflicts in the switch. The point of this section is the computation of D. The factor of 2 comes from the fact that each request involves two messages through the switch (one each way), and therefore there are two opportunities for delay. D can be expressed in terms of the probability of collision (q) on a single try through the switch as follows:

$$D = q(D_0 + D) \tag{2.3}$$

where D_0 is the time lost in the trial that failed due to contention. D_0 tends to be 2 microseconds. This simple equation (2.3) ignores two effects. First, the time lost in the trial is not a constant since



Contention Delay As A Function Of Load
Figure 2.1

we deliberately make it a random variable. Second, successive trials are not independent, particularly when the random variable happens to be small, since it is more likely that you will run into the same message when you simply try again. A more detailed examination of these issues convinces us that the simple equation is correct, but that the value of D_0 is some 10 to 20% larger than one would calculate by averaging over the random variable. We include all of this in the 2 microseconds allowed for D_0.

Rewriting equations (2.2) and (2.3) we get:

$$D = D_0 [q/(1-q)] \tag{2.4}$$

$$F = E + R + 2D_0 [q/(1-q)] \tag{2.5}$$

The next step is to express q as a function of the probability (p) of collision at a single stage of the switch. This function will depend on the details of the switch geometry, but the largest effect will be from the number of stages or ranks (r) in the switch. If the switch is fully configured (i.e. the number of processors is a power of four), it is easy to write the expression for q: the chance of overall success is the product of the chances of individual success:

$$1 - q = (1 - p)^r \tag{2.6}$$

If the switch is not fully configured, then the probability of collision at the different stages of the switch is not the same, and the corresponding equation becomes:

$$1 - q = (1 - p_1)(1 - p_2) \cdots (1 - p_r) \tag{2.7}$$

In the rest of this note we will assume fully configured switches. Since $r \leq 4$, it is conceivable to carry out the computation using these expressions.

Since both p and q are relatively small, it is a good approximation to say that:

$$q = r * p \tag{2.8}$$

Let us now examine p, which will depend both on the geometry and the load. If the switch is fully configured, so that the load on the input of a single switch element is fully balanced, then:

$$p = (3/4)\,(R/F) \tag{2.9}$$

where R is the 4μsec roundtrip time a message occupies the switch. Since F is the time between message starts, R/F is the probability that any particular switch path will be in use. The factor 3/4 comes from the fact that at the time the path is requested, three other paths (in the 4 x 4 Butterfly Switch implementation) can be delivering data with a probability R/F, and this data is distributed over four output paths.

Let us express equation (2.8) as:

$$q = k\ R/F \tag{2.10}$$

where k = 3 r/4. Here k depends only on the geometry of the switch since r is the number of ranks, and is a number between 1.5 and 3.0.

We now have two equations which represent our model of the contention delay in the switch:

$$F = E + R + 2D_0\,[q/(1-q)] \tag{2.5}$$

and

$$q = k\ R/F \tag{2.10}$$

This can be solved directly as:

$$F = E + R + kR + \frac{\sqrt{(E + R - kR)^2 + 8KRD_0}}{2} \tag{2.11}$$

We seek the relative slowdown x due to contention relative to the time for a cycle in the absence of contention. If we set A equal to the time in the absence of contention A = E + R; then:

$$\text{relative slowdown } x = \frac{(F - A)}{A} \tag{2.12}$$

now;

$$x = \frac{1}{2}\left(- \frac{kR}{A} \right)\left(\sqrt{1 + \frac{8kRD_0A}{(A-KR)^2}} - 1 \right) \tag{2.13}$$

We note that as A becomes larger, the slowdown due to contention approaches zero, as it should. Equation (2.13) is plotted as a function of E, the size of the inner loop, in Fig. 2.2.

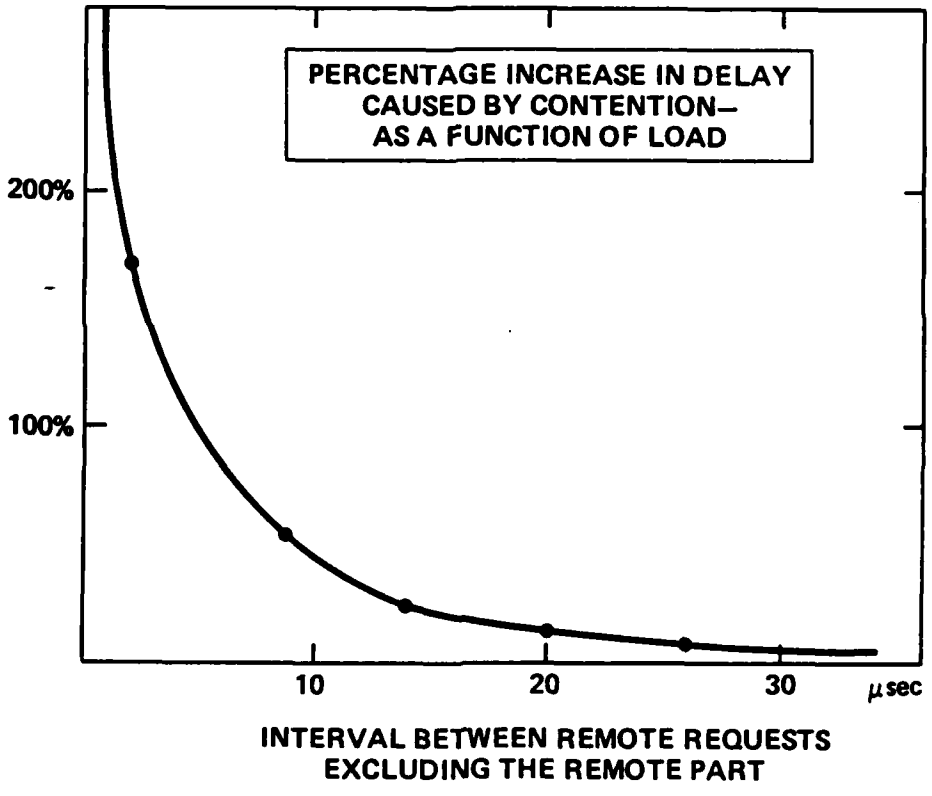
The various parameters corresponding to the configuration of the 128-Processor Test Bed are: R = 4, D_0 = 2, and k = 3. Figure 2.2 illustrates that the slowdown due to contention decreases rapidly as the length of the inner loop E increases. By the time E is 10 microseconds, the slowdown is 40%; and at E of 25 microseconds the slowdown is only 4 percent.

One might wonder why the slowdown is not infinite when the inner loop becomes zero. This is because there is at least the four microsecond delay due to the serializaton of the remote reference in every loop. Here we are calculating only the contention delay, not the total delay due to the Butterfly Switch. The total delay is about 4 microseconds even when E approaches zero.

2.3. Conclusions

Equation 2.13 represents a model of contention slowdown in the Butterfly Switch based upon the specifics of the switch and the average interval between remote references in the program. For programs where the interval between remote references is of the order of 20 microseconds, the contention delay is dominated by the simple serial delay across the switch. Since typical MC68000 instructions require about 2 microseconds, the balance between processor and switch design is good.

In the next section, experimental data is presented for switch contention.



Percentage Increase In Delay Caused By Contention
Figure 2.2

3. SWITCH CONTENTION MEASUREMENT RESULTS

3.1. Introduction

This section documents a performance test of a 16-node Butterfly system. Several experiments were performed, all of which involved a test program that performed a matrix multiplication operation on a pair of 50 by 50 matrices. We designed this series of experiments to verify two theories: one that predicts hardware contention in the Butterfly switch, and another that addresses the effective utilization of the processors. Both theories suggest that this should be an "easy" problem for the Butterfly system and that we should be able to make efficient use of a large number of processors on a matrix multiplication problem of any reasonable size.

These experiments are not intended to test the arithmetic capabilities of the Butterfly machine, which are currently somewhat restricted due to the lack of floating point hardware (Note: this will be remedied in late 1985). In all of the experiments described, the test program operates on the elements of two input matrices in such a way that the pattern of memory references and program steps is what one would see if one were doing a matrix multiply. In the first two tests, the program actually does the multiplication. In the third test, we replace all multiplication instructions by additions. This produces nonsensical answers, but it gives a useful indication of what the performance of the Butterfly system would be if it had a fast hardware floating point unit. It also provides a severe test for switch contention.

3.2. The Program

As noted above, the test program was very nearly the same for all experiments. We chose the calculation of a single element of the result matrix as the basic unit of work. It would have been

just as useful to choose a whole row of the answer or the sum of only 10 terms of a single element. These would have yielded different experiments but much the same results.

The test program uses utilities from the Uniform System reference written by Will Crowther for the Butterfly machine. The generator primitive "HelpOnArray" is used to distribute the workload. The primitive "AllocateScatterMatrix" is used to allocate space for the operand and result matrices.

"HelpOnArray" is a processor allocation routine, which schedules processors from a system-wide pool of idle machines to a set of tasks defined by a pair of indices. In these tests, a unit of work is defined by the row and column number of the element in the output matrix, to be computed. The generator is implemented by a pair of counters, which are incremented using microcode-supported atomic Add-to-Memory primitives, provided by the operating system.

"AllocateScatterMatrix" is a storage allocation routine that allows a program to store a matrix "by rows" or "by columns". With the first option, each row of the matrix in question is stored on some random processor; with the second option, each column is stored on a different processor. In reality, the allocator does not try to be random. It simply walks through the memories of all the available processors in ascending order.

With either option, the allocator spreads the elements of the matrix evenly among the memories in the machine in order to minimize contention for any particular memory. We make no attempt to couple the storage and processor allocators; as a result a processor will usually be working on data in another processor's memory. We do this deliberately to insure that there is switch contention to measure. The choice of the row or column options has no impact on switch or memory contention; the purpose is to simplify the effort required to step through successive elements of a row (or column) in the array.

The values of the input matrix elements were generated by a small program. One matrix featured consecutively increasing entries, and the other was a diagonal matrix with all diagonal elements being 3. This made it possible to verify, by inspection, that the program was computing correct results. The fact that many elements were zero had no effect on the running time of the program, since neither the hardware nor the software treats zero as a special case.

A subjective result is that the test program was relatively easy to write. The entire program fits on a single page, and more than half of the code is devoted to setting up the initial matrices and printing out the final results. One successful aspect of the debugging effort was the use of the VAX-based simulator for the Uniform System. The application code was fully debugged in the uniprocessor environment of the VAX.

3.3. System Overhead

The Chrysalis[TM*] operating system and the Uniform System that runs on top of it impose very little per-processor overhead. With the exception of the processor that is connected to the console terminal, Chrysalis background processes have been shown to account for less than 3.5% of the cycles on any Processor Node. On the processor connected to the console, there is an additional process that polls the terminal line, using an additional 2% of the cycles. Since the Uniform System creates only one application process per processor node, the amount of time lost due to context switching is also very small. The only Chrysalis primitives used to any extent by the Uniform System are Dual Queue operation and atomic add-to-memory operations. Both of these are supported by microcode, and are quite fast.

*Chrysalis is a trademark of Bolt Beranek and Newman Inc.

3.4. Results

These early experiments were conducted in January 1985 with a 16-processor Butterfly machine using a 2-rank switch. Further experiments will be performed with 128 processors and a 4-rank switch. The programs were coded and debugged on a VAX 780, and then loaded into the Butterfly machine. Once loaded, the program was run from the Butterfly console.

For each experiment we ran a test program on a series of configurations ranging from one to sixteen processors. We measured and recorded the execution time for each trial. In each trial the allocation of memory remained the same, so that we ran the program with the processor part of some nodes turned off but the memory part turned on. Although this method is somewhat different from measuring a series of machines each with a different number of processor boards, it is the best way to measure the effect of contention. We repeated each set of trials several times. There was little variation in the results from trial to trial, and the execution times varied quite smoothly as a function of the number of processors. This suggests that there were no artifacts present introducing noise into the results.

There are many ways to present the results of this kind of experiment. The most straightforward way is to plot the number of effective processors in use vs. the number of actual processors in use. If all the processors are being used with 100% efficiency, the resultant plot should be a straight line. This kind of plot is included for each of the experiments performed here. Since, in many of the experiments the deviation from the ideal straight line is small, we also present the percentage deviation from the ideal straight line in tables.

For each experiment then, we have a plot of effective processors vs actual processors, plus a four column table showing the raw data (time in clock ticks), the time in seconds, the number of effective processors, and the percentage deviation from a line of 100% efficiency. Usually the table

will extend from 1 to 16 processors (although we made some of the measurements on a machine missing two processors). Occasionally the percentage deviation will be slightly negative, showing a better than linear speedup. As explained elsewhere in the test, this anomaly is due to the fact that the first processor has an extra task relative to the rest, since it must maintain the keyboard and display. The effect is in fact real, and two processors do slightly more than twice the work of one processor where the one is connected to the console. The effect is also small, around 2 to 3 percent, and should be ignored.

We based the reported times and efficiencies on measurements of the matrix multiplication itself. The setting up the operand matrices and the printing of the results are not included in the measurements. The overhead necessary to capture the attention of the extra processors and to direct them to start work on the multiplication is included in the measurements.

Note that the Butterfly Switch introduces two sources of delay: one is the fixed cost of a remote reference, which is due entirely to the time that it takes to clock a read request, and the corresponding reply through the switch. The other is contention delay, which is a function of the average number of times that a message fails to reach its destination because some other message has blocked its way. The current series of experiments is devoted to measuring the second kind of delay – contention delay, and comparing it to the theoretical predictions derived in Section 2. The first kind of delay can be calculated precisely from microcode instruction counts, and is 4 microseconds.

The efficiency quoted here is therefore the EXTRA loss attributable to conflicts, after the fixed cost of a remote reference in the absence of contention has been added. There are other measures of efficiency that are just as useful (as long as their definitions are clearly stated). We chose this one because we found it easy to relate to the actual performance application software running on the Butterfly Parallel Processor.

3.4.1. Experiment 1: Floating Point Matrix Multiply

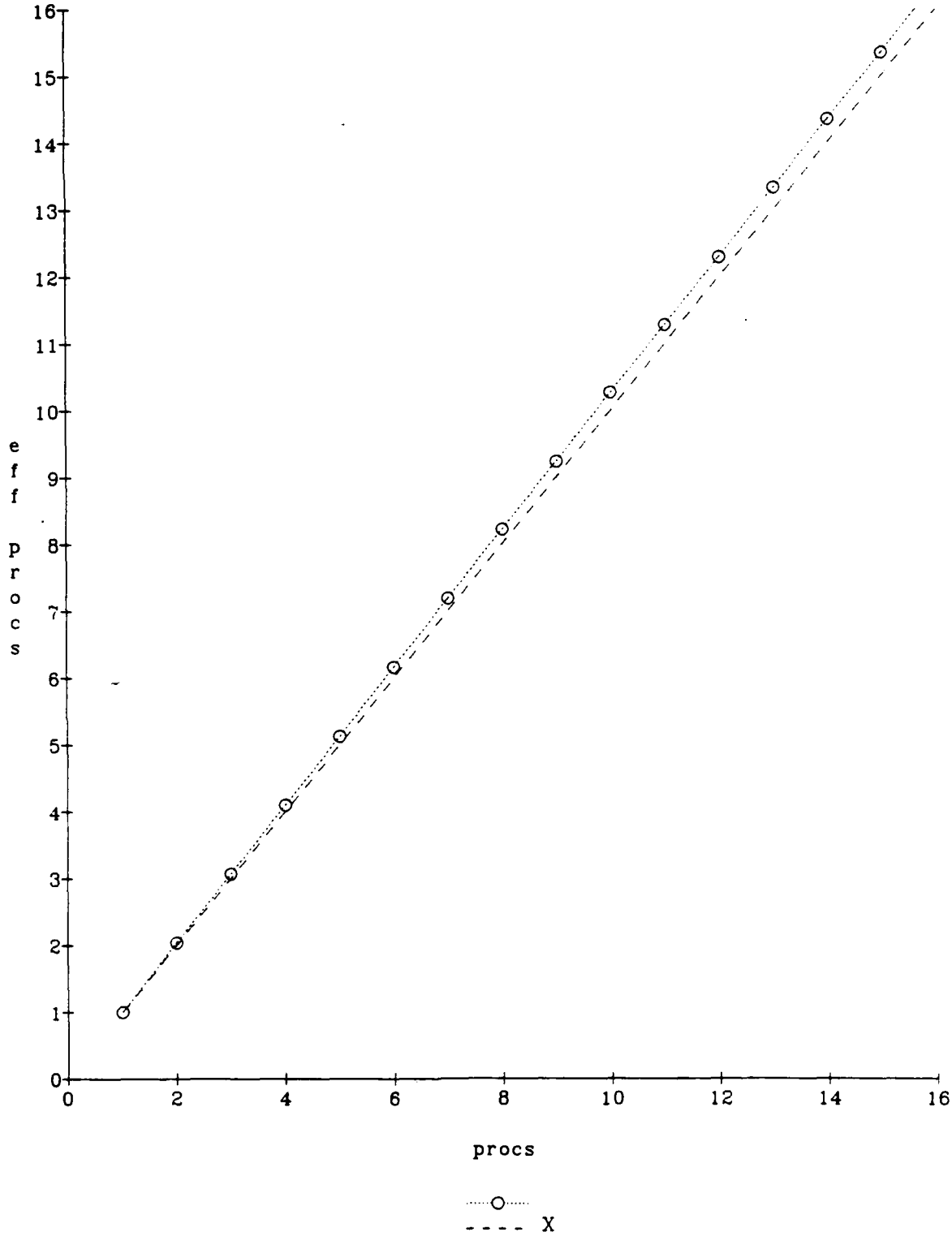
In this experiment, two 50x50 matrices of floating point numbers were multiplied together, using the methods described previously. Table 3.1 shows that the per processor efficiency is essentially constant (at 1) for all configurations. This means that the program achieves linear speedup over the whole experiment. Looking at the tiny variations in the data leads to the surprising conclusion that efficiencies are in fact slightly greater than one. This due to the fact that the "baseline" experiment, which established the value of T(1) was run in the processor connected to the console. As noted earlier, this processor is subject to slightly greater system overhead than the rest of the processors in the system. Figure 3.1 shows the same results in a different form by plotting the number of actual processors versus the number of effective processors for each configuration.

A simple but useful measure of the load on the Butterfly Switch is the frequency with which a single processor makes remote references while executing the inner loop of the test program. This number can be deduced from the time a single processor takes to complete the problem and the total number of external references required to complete the problem. There are 2500 elements in the result matrix and 100 references to the operand matrices for each dot product, so the total number of references to do the multiply is 250,000 (plus another 3000 or so for setup and storing the results). Since a single processor requires about 100 seconds to complete the problem, the time between memory references is about 400 microseconds. Each of these references is a double word reference that requires 8 microseconds of "switch time."

	Clock Ticks	Total Time (sec)	Effective Processors*	Deviation (%)
1	1618572	101.160750	1.000000	0.000000
2	794906	49.681625	2.036180	-1.809019
3	527552	32.972000	3.068080	-2.269350
4	394706	24.669125	4.100703	-2.517570
5	315607	19.725437	5.128441	-2.568828
6	262630	16.414375	6.162936	-2.715608
7	225224	14.076500	7.186499	-2.664268
8	196877	12.304813	8.221235	-2.765432
9	175265	10.954062	9.234998	-2.611094
10	157696	9.856000	10.263875	-2.638748
11	143475	8.967187	11.281213	-2.556480
12	131686	8.230375	12.291147	-2.426226
13	121478	7.592375	13.323993	-2.492252
14	112809	7.050562	14.347898	-2.484984
15	105504	6.594000	15.341333	-2.275554
16	98831	6.176937	16.377169	-2.357307

* (normalized to the task running on the first processor using remote memory)

Floating Point Matrix Multiply 2-Column Switch
Table 3.1



Floating Point Matrix Multiply
Figure 3.1

3.4.2. Experiment 2: Fixed Point Matrix Multiply

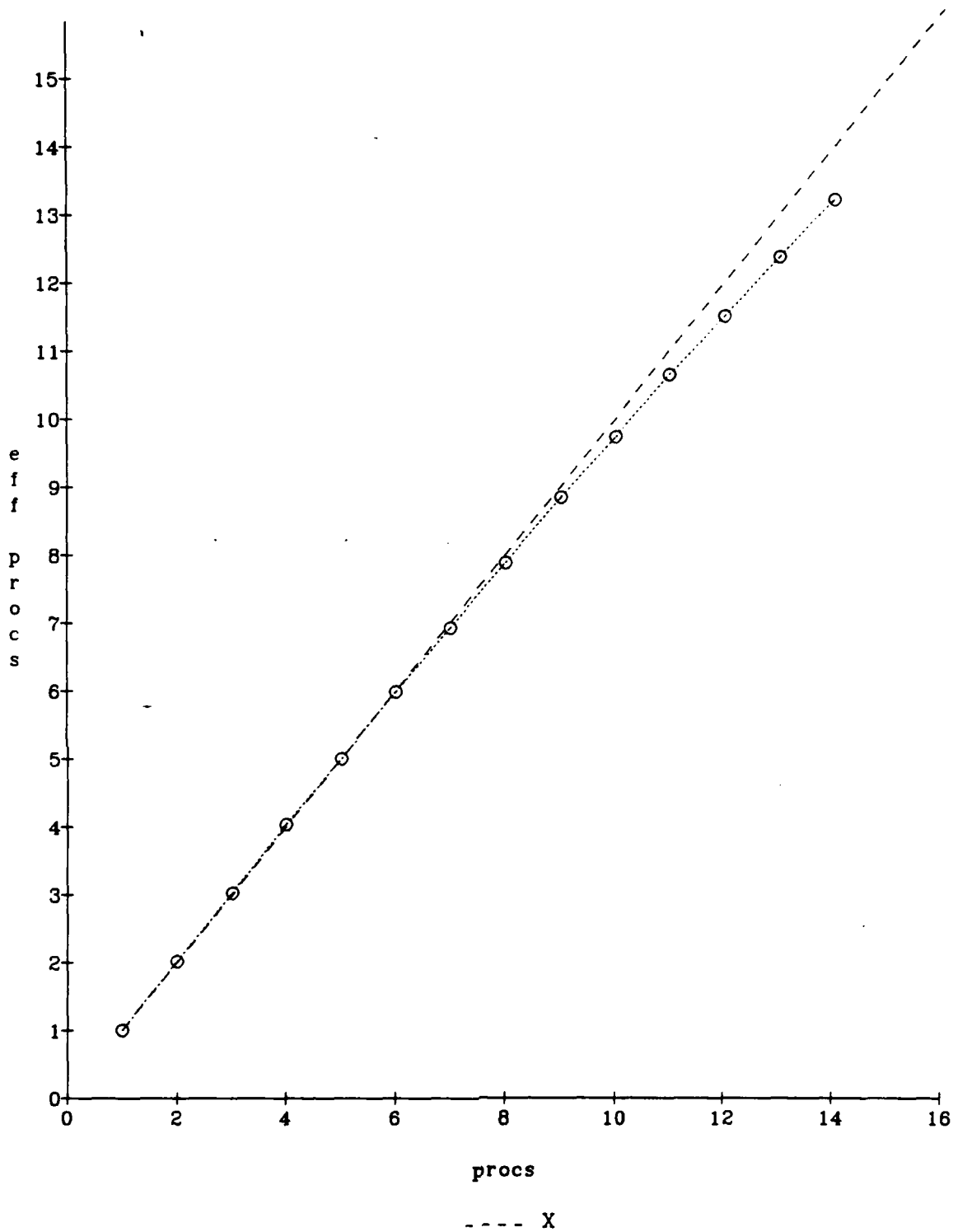
In this experiment, the same computation was done using 32-bit integer arithmetic. For real problems this represents marginal precision, since numbers larger than about 5000 lead to overflow. However, that is not important for the purposes of the experiment. Table 3.2 shows that after an expected initial rise of 1%, the efficiency falls gently to 95% at 14 processors. The runtime was 15.4 seconds for the single processor case, which works out to roughly 62 microseconds between references. We are still not using the switch heavily.

From the results of this experiment, the rate of conflict in the switch appeared to grow nearly twice as fast as predicted by the theoretical model described in Section 1. Examination of the test program indicated a systematic cause for this growth. The task generator hands out elements of the result matrix in ascending order (column 1, row 1; column 1 row 2; etc). This means that some of the columns in one of the operand matrices are being accessed very frequently, while the rest of the columns in that matrix are not being accessed at all. To see whether this was the cause of the problem, the test program was modified to compute element (i, (i+j) mod 50) when given the coordinates (i, j) to spread references to the operand matrices more evenly across the machine.

The results from this version of the experiment are shown in Table 3.3 and Figure 3.2. The qualitative behavior is the same, but the efficiency now increases to 97% instead of 95%, which matches the expected results predicted in Section 2.

	Raw Data	Time (sec)	Effective Processors	Deviation (%)
1	246434	15.402125	1.000000	0.000000
2	122502	7.656375	2.011673	-0.583664
3	81517	5.094812	3.023099	-0.769983
4	61202	3.825125	4.026568	-0.664194
5	49281	3.080062	5.000588	-0.011769
6	41183	2.573938	5.983877	0.268719
7	35597	2.224812	6.922887	1.101618
8	31278	1.954875	7.878829	1.514643
9	27884	1.742750	8.837828	1.801909
10	25335	1.583437	9.727018	2.729820
11	23164	1.447750	10.638663	3.284878
12	21429	1.339312	11.500023	4.166472
13	19918	1.244875	12.372427	4.827485
14	18655	1.165937	13.210078	5.642302

Fixed Matrix Multiply Without Skew
Table 3.2



Fixed Point Matrix Multiply
Figure 3.2

3.4.3. Experiment 3: High Switch Utilization Test

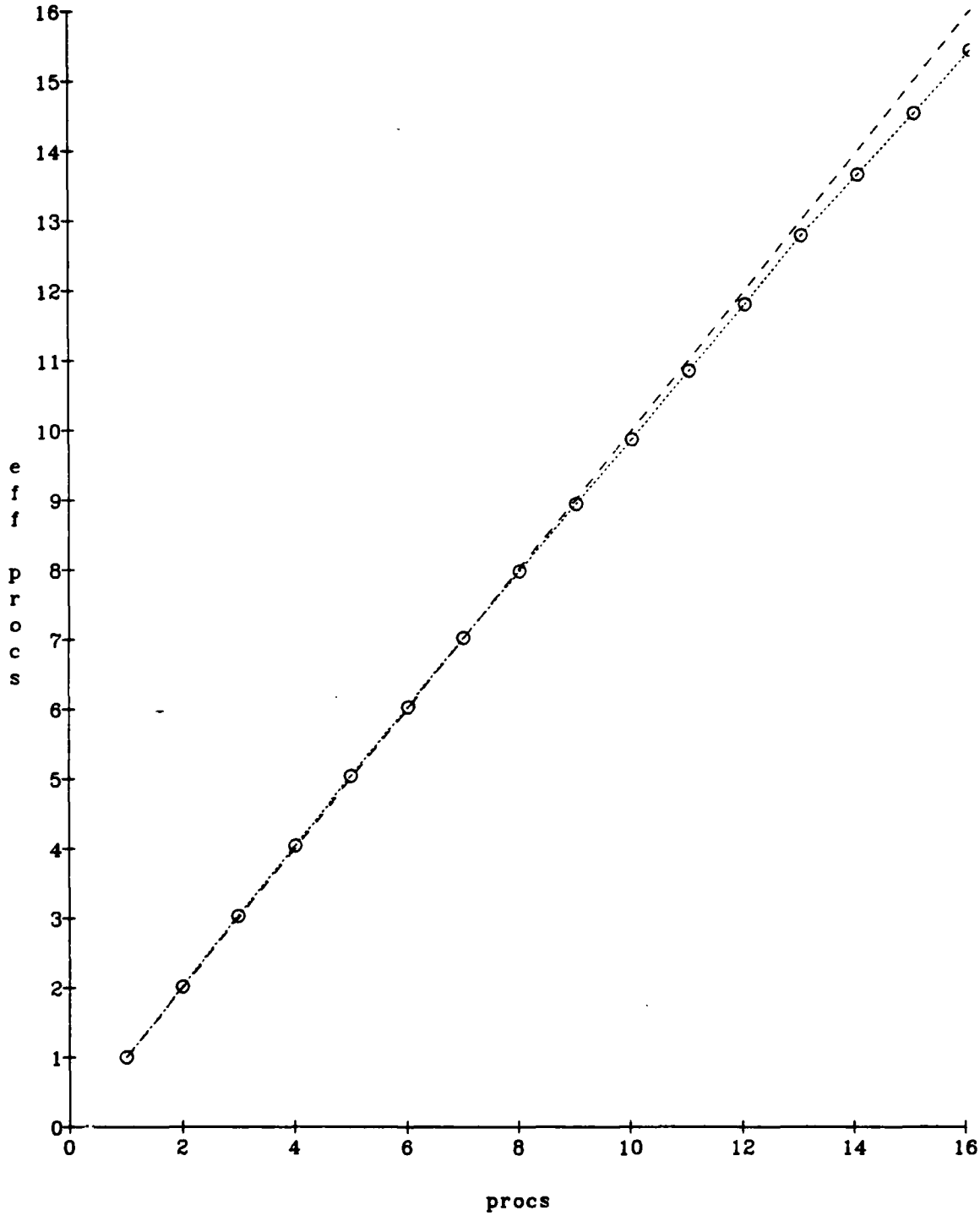
In this experiment, the multiply instruction of the dot product was replaced by an add instruction. This produces a nonsense answer, but it does so much faster than the multiply -- greatly increasing the traffic through the switch. It is also a reasonable simulation of hardware floating point. In a slightly different version of this experiment, we tuned the inner loop of the test program by declaring heavily used local variables to be register variables. In all previous experiments this fine tuning would have caused little change in the results. In this version of the test program, however, the inner loop was so short that the tuning made a difference. We avoided systematic contention for rows and columns of the operand matrices by using the technique described in the previous experiment.

Table 3.4 shows the result of untuned torture test where the efficiency falls almost linerarly to 83% at 14 processors. The runtime was 7 seconds for the single processor case, which works out to roughly 27.5 microseconds between references.

When the tuning was done, the interval between remote references in the inner loop was 1.7 microseconds. Since it takes 8 microseconds to fetch a 32-bit word across the Butterfly Switch in the absence of contention, each processor is spending half of its time making references through the switch in this experiment. As shown in table 3.5, the efficiency drops off to 65% in a 16-processor configuration.

	Raw Data	Time (sec)	Effective Processors	Deviation (%)
1	247920	15.495000	1.000000	0.000000
2	122973	7.685812	2.016052	-0.802615
3	81795	5.112187	3.030992	-1.033070
4	61363	3.835187	4.040220	-1.005492
5	49222	3.076375	5.036772	-0.735444
6	41165	2.572813	6.022592	-0.376553
7	35361	2.210062	7.011114	-0.158771
8	31151	1.946937	7.958653	0.516837
9	27765	1.735312	8.929227	0.786362
10	25135	1.570937	9.863537	1.364631
11	22873	1.429562	10.838980	1.463814
12	21017	1.313562	11.796165	1.698625
13	19403	1.212687	12.777406	1.712265
14	18159	1.134937	13.652734	2.480470
15	17071	1.066937	14.522875	3.180833
16	16077	1.004812	15.420787	3.620078

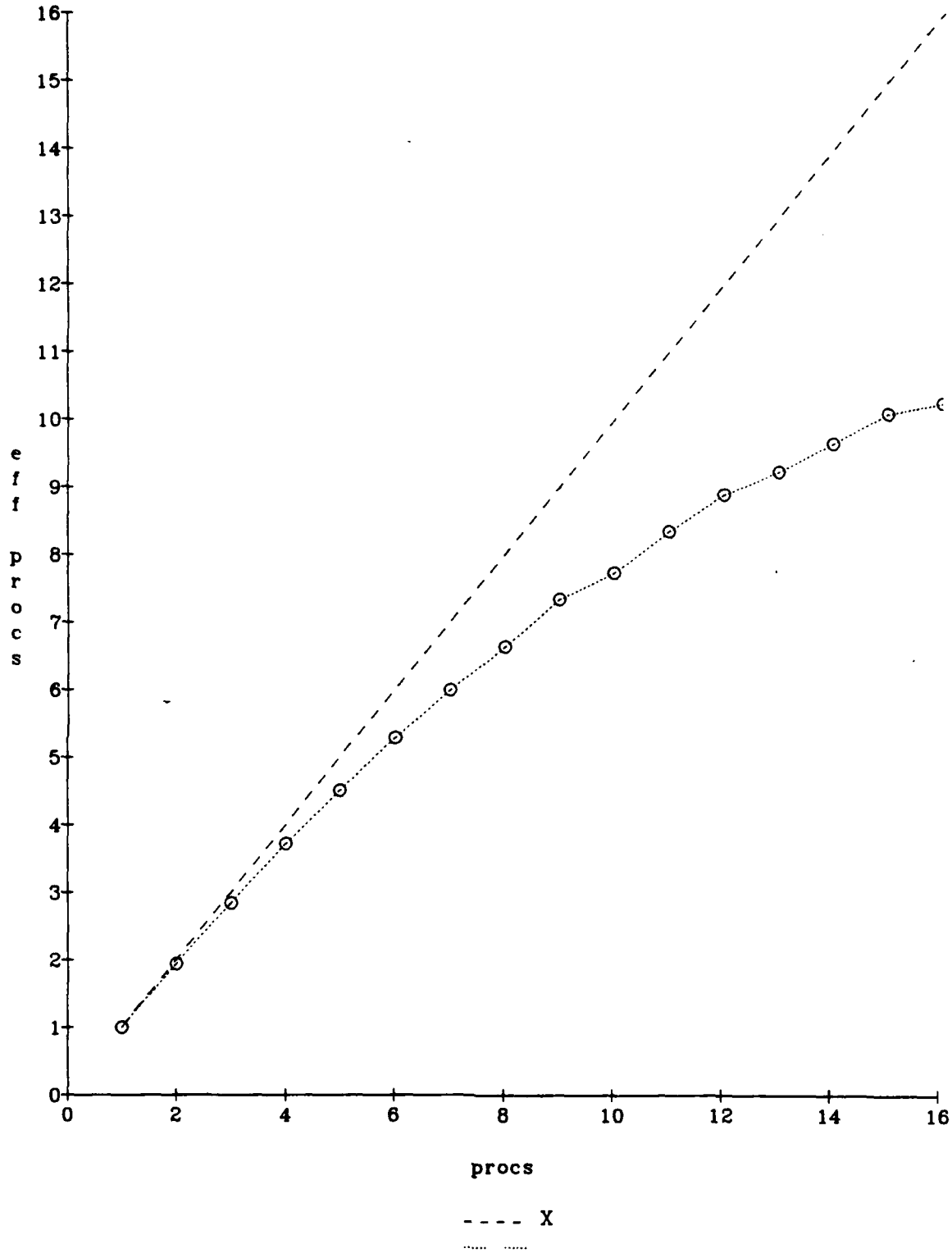
Fixed Matrix Multiply 2-Column
Table 3.3



Simulated Floating Point Matrix Multiply
Figure 3.3

	1-Raw Data	2-Time (sec)	3-Effective Processors	4-Deviation (%)
1	67596	4.224750	1.000000	0.000000
2	34908	2.181750	1.936404	3.179787
3	23792	1.487000	2.841123	5.295898
4	18173	1.135812	3.719584	7.010400
5	15003	0.937687	4.505499	9.890022
6	12792	0.799500	5.284240	11.929331
7	11280	0.705000	5.992553	14.392097
8	10191	0.636937	6.632911	17.088608
9	9204	0.575250	7.344198	18.397798
10	8747	0.546687	7.727907	22.720933
11	8105	0.506563	8.340037	24.181482
12	7606	0.475375	8.887194	25.940047
13	7325	0.457813	9.228123	29.014439
14	7011	0.438187	9.641421	31.132710
15	6705	0.419062	10.081432	32.790455
16	6601	0.412563	10.240267	35.998334

Matrix Multiply Torture Test
Table 3.4



Matrix Multiply Torture Test
Figure 3.4

4. CONCLUSIONS

The experiments described in this report explore the parallel use of the Butterfly Parallel Processors, measure contention in the Butterfly Switch, and prepare for measurements of the 128-Processor Test Bed with a 4-rank switch. They also validated the switch contention model for a 2-rank switch.

As indicated by the results of the experiments, parallel processors work effectively on the matrix-multiply task. In effect all inefficiencies noted were due to switch contention. We did see one "performance bug" when the program naively computed elements in a simplistic order. This introduced a systematic coupling between the processor allocation and the storage allocation, causing a minor performance degradation. By introducing a little more randomness into the order of calculations, we confirmed switch contention to be the key inefficiency.

In the matter of contention, the Butterfly machine performed as predicted by the model developed in Section 2. When the interval between references was 400 microseconds, switch contention had essentially no effect. As we increased the load on the switch, switch contention took a gradually increasing toll. At the extreme of the experiments (the "torture" test of experiment 3), where a memory reference occurred every 17 microseconds, the sixteen processors performed as ten effective processors. More typical results yielded fifteen effective processors out of sixteen.

It is difficult to write a useful program on the MC68000 with a smaller loop than the one used in Experiment 3; this gives an upper bound on the efficiency loss due to contention in a Butterfly system with a two-rank switch. It is also easy to expand the switch to more ranks with parallel paths for sixteen processors to substantially lower the contention over that measure.

It should also be noted also that we designed these programs to create switch contention so that it could be measured. An efficient matrix multiply program, using block transfers and cached rows, can achieve contention rates of less than 5% in the simulated hardware case, where the interval between remote 32-bit references is about 9μsec. We are now preparing to repeat these experiments with a larger machine (128 processors) and a four-rank switch. These results will be reported in a future report.

END

DTIC

5-86